(12) UK Patent Application (19) GB (11) 2 345 360 (13) A

(43) Date of A Publication 05.07.2000

(21) Application No 9923193.8

(22) Date of Filing 30.09.1999

(30) Priority Data
(31) 332146 (32) 02.10.1998 (33) NZ

(71) Applicant(s)
Fisher & Paykel Limited
(Incorporated in New Zealand)
78 Springs Road, East Tamaki, Auckland,
New Zealand

(72) Inventor(s)
Scott Bevin Cornwall
Anne Marie Philpott
Tony James Bryant
Derek Ward

(74) Agent and/or Address for Service
Forrester Ketley & Co
Forrester House, 52 Bounds Green Road, LONDON,
N11 2EY, United Kingdom

(51) INT CL[7]
G06F 9/44

(52) UK CL (Edition R)
G4A APL

(56) Documents Cited
US 5247693 A

(58) Field of Search
UK CL (Edition R) G4A APL
INT CL[7] G06F
ONLINE:WPI,EPODOC,JAPIO

(54) Abstract Title
Programming programmable controllers

(57) A programming environment for programming a control processor generally implements the methods set forth in the IEC 1131-3 standard. The environment includes at least the provision of a graphically representable function block based top-level structure. The environment provides a specialised state-diagram function block type, adapted to implement one or more state-diagrams defined by a user.

GB 2 345 360 A

This print takes account of replacement documents submitted after the date of filing to enable the application to comply with the formal requirements of the Patents Rules 1995

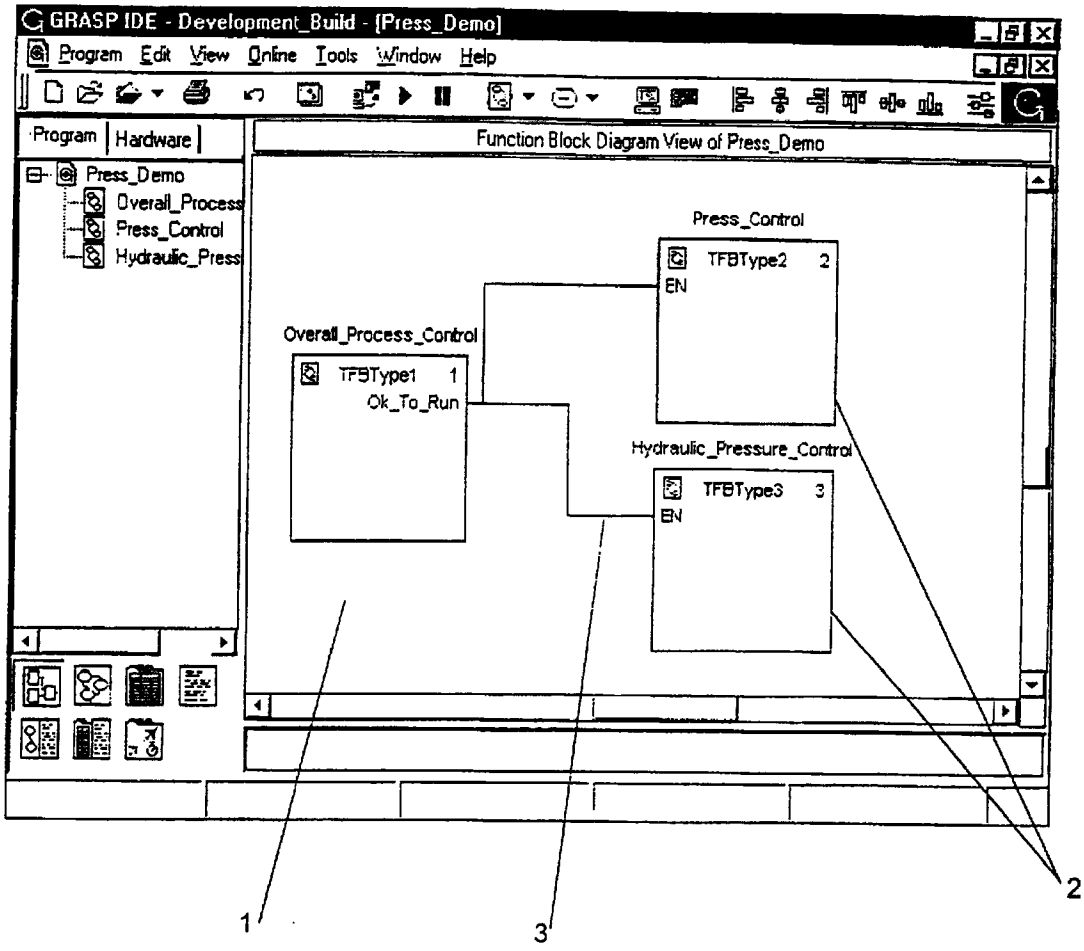

GRASP IDE - Development_Build - [Press_Demo]
Program Edit View Online Tools Window Help
Program Hardware
Press_Demo
Overall_Process
Press_Control
Hydraulic_Press
Function Block Diagram View of Press_Demo
Press_Control
TFBType2 2
EN
Overall_Process_Control
TFBType1 1
Ok_To_Run
Hydraulic_Pressure_Control
TFBType3 3
EN
1
3
2

FIG. 1

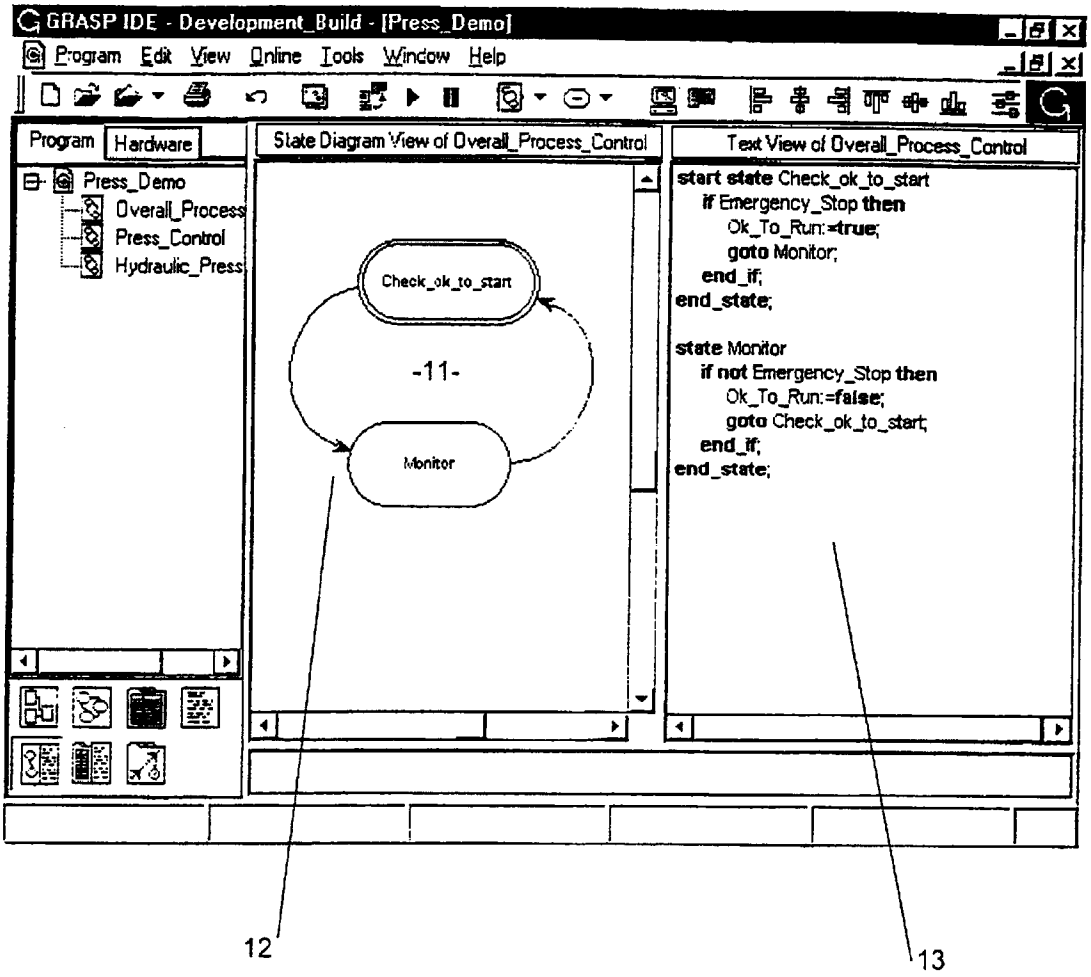

GRASP IDE - Development_Build - [Press_Demo]
Program Edit View Online Tools Window Help
Program
Hardware
Press_Demo
Overall_Process
Press_Control
Hydraulic_Press
State Diagram View of Overall_Process_Control
Check_ok_to_start
-11-
Monitor
Text View of Overall_Process_Control
start state Check_ok_to_start
if Emergency_Stop then
Ok_To_Run:=true;
goto Monitor;
end_if;
end_state;
state Monitor
if not Emergency_Stop then
Ok_To_Run:=false;
goto Check_ok_to_start;
end_if;
end_state;
12
13

FIG. 2

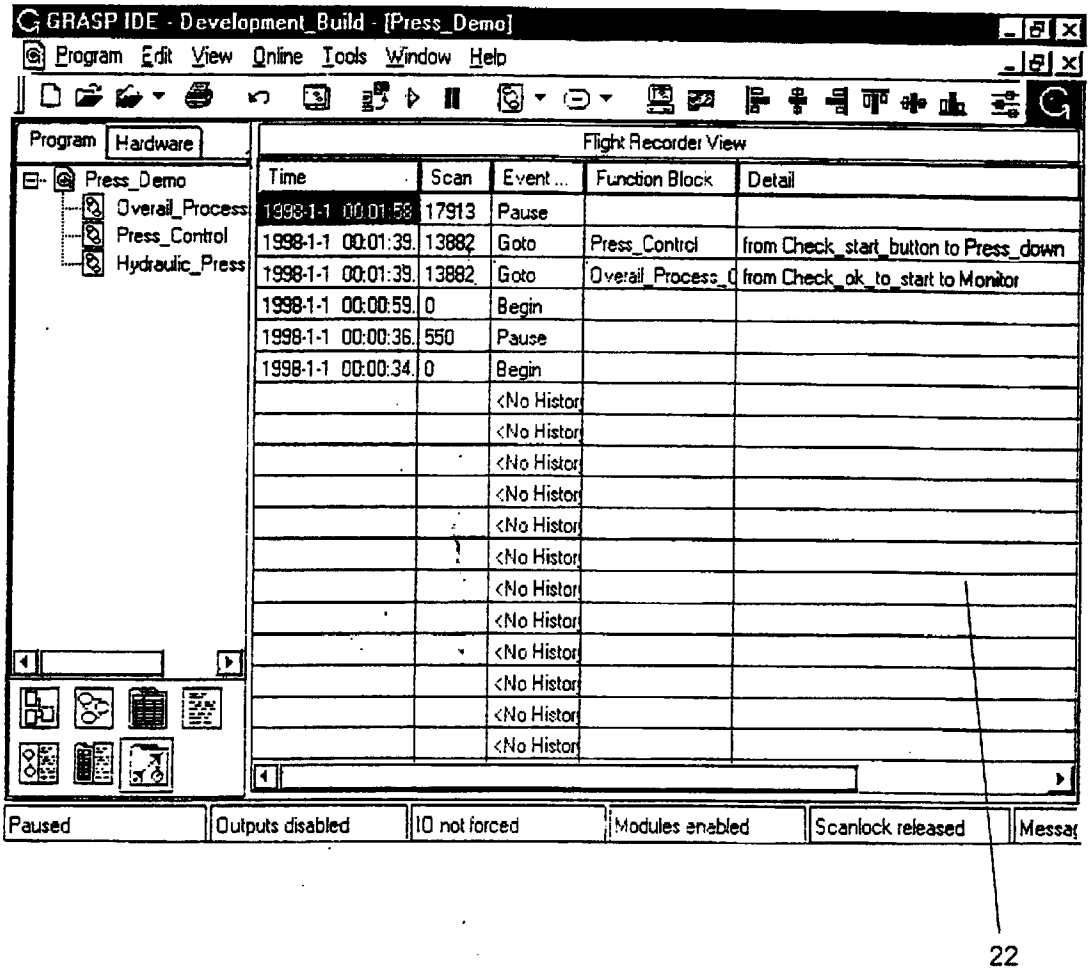

GRASP IDE - Development_Build - [Press_Demo]
Program Edit View Online Tools Window Help
Program
Hardware
Press_Demo
Overall_Process
Press_Control
Hydraulic_Press
Flight Recorder View
Time
Scan
Event ...
Function Block
Detail
1998-1-1 00:01:58. 17913 Pause
1998-1-1 00:01:39. 13882 Goto Press_Control from Check_start_button to Press_down
1998-1-1 00:01:39. 13882 Goto Overall_Process_C from Check_ok_to_start to Monitor
1998-1-1 00:00:59. 0 Begin
1998-1-1 00:00:36. 550 Pause
1998-1-1 00:00:34. 0 Begin
<No Histor
Paused
Outputs disabled
IO not forced
Modules enabled
Scanlock released
Messa
22

FIG. 3

PROGRAMMABLE CONTROLLERS AND APPARATUS FOR THE PROGRAMMING THEREOF

FIELD OF THE INVENTION

This invention relates to programmable controllers and to apparatus for the programming thereof.

BRIEF DESCRIPTION OF THE PRIOR ART

In US Patent 4802116, we taught how a programmable controller can be programmed in terms of a state diagram paradigm. Since that time, many developments have taken place in the field of control systems, particularly in the way that programs are entered and edited.

A standardised programming system has been developed by the international community and formalised as IEC Standard 1131-3. This is desirable because standardising approaches to using complex systems simplifies the problems that occur when personnel are required to move from one type of system to another. Retraining is reduced and fewer errors occur.

IEC 1131-3 has been designed to provide a standardised framework for PLC or control computer program design and use. It is also designed to allow latitude for ongoing innovation by those companies marketing PLCs. A control computer, in this context, means a computer used for the type of task handled by a PC running control software generally considered to follow IEC 1131-3 principles.

Various programming methods are standardised. These include Instruction List, Structured Text, Ladder Logic, and Function Block. Sequential Flow Charts are provided for defining sequential processes. The various methods are used with varying popularity in different parts of the world.

Programming based on a State Diagram paradigm is not provided. It can be reasonably argued that IEC 1131-3 is already too complex by virtue that it provides several methods of program preparation where one would have sufficed. It can also be argued that it is more beneficial to be able to use State Diagrams than Sequential Flow Charts, or at the least, to be able to do so offers the public a useful choice.

One problem is therefore to find a way of providing State Diagrams, to be able to do so within the spirit of IEC 1131-3 and at the same time to avoid adding any more complexity than is essential.

OBJECT OF THE INVENTION

Accordingly it is an object of the present invention to provide a programmable controller and/or apparatus for the programming thereof which goes some way toward meeting the abovementioned objectives or which will at least provide the public with a useful choice.

SUMMARY OF THE INVENTION

In a first aspect the invention consists in apparatus for assisting a programmer in the programming of a programmable controller, comprising:

means for representing a function block diagram having one or more function blocks represented graphically, each with inputs and/or outputs which inputs or outputs may be connected graphically to outputs or inputs of another said function block,

means for allowing user creation and manipulation of said function blocks and of said input/output connections,

means for representing a state diagram, having one or more states represented graphically, and possible transitions between states represented graphically by connections between states,

means for allowing user creation and manipulation of said states and of said transitions,

means for allowing a user to enter coded instructions in relation to a state, and

means for compiling program code for execution by a control processor from said user created function block diagram, state diagrams and entered instructions.

Preferably said apparatus includes means for internally associating each said state diagram with a said function block and wherein said compiling means reports an error if said user has entered instructions or a transition between states of a state diagram which requires an input or sets an output which is not represented in the function block diagram as being an input or output from the respective function block.

Preferably said apparatus includes means for representing graphically one of said states as a starting state.

Preferably said apparatus includes means for debugging which analyse a recorded history of operation of a control processor programmed with said compiled program code, and include means for displaying state activity changes which indicate on said graphically represented state diagrams the route taken by the state activity during said recorded operation of the control processor through the network formed by the states and the

transitions between the states.

Preferably said means for compiling program code compiles said code into individually identified code blocks, one block for each state of each said state diagram, and said apparatus includes downloading means for downloading one or more individual blocks into said control processor for the purpose of adding new states or of modifying existing states.

Preferably said downloading means downloads said code blocks in the period between program scans, and changes the value in an identifying variable indicating the block of code which represents a state in the state diagram, and in this way causing the controller to execute different code for the state when it is active after the program swap than would have been executed prior to the program swap.

In a further aspect the invention consists in a programming environment for programming a control processor, said environment generally implementing the methods set forth in the IEC 1131-3 standard, including at least the provision of a graphically representable function block based top-level structure, characterised in that the environment provides a specialised state-diagram function block type, which function block type is adapted to implement one or more state diagrams defined by a user.

Preferably said function blocks representing state diagrams have an associated state variable for each state diagram, provided by said programming environment, the value of which is indicative of the state that is currently active on the state diagram.

Preferably said function blocks representing state diagrams have a status variable, provided by said programming environment indicating at least whether said state diagram is running or stopped.

Preferably said function blocks representing state diagrams have a variable, provided by said programming environment, indicating whether the current scan is the first scan since the current state, as defined by said state variable, has become active since its last period of inactivity.

Preferably said state variable is a pointer or an index into a table.

Preferably said state variable is used as part of an automatic operating system function that records the history of state activity as the control process progresses.

Preferably said state diagram is coded in one of the standard IEC 1131-3 languages with the addition of state and transition defining functionality.

Preferably the state functionality is included by virtue of the operating system storing state code blocks as individually identifiable items, referenced by said state variable.

Preferably said programming environment compiles said state diagram related code

into individually identifiable blocks, with one said block per state, and said identifiable blocks may be individually loaded into the controlling process for the purpose of adding new states, or modifying existing ones in the period between program scans, and implements the program swap by changing the value in an identifying variable indicating the block of code which represents a state in a state diagram, and in this way causes the controller to execute different code for the state when it is active after the program swap than would have been executed when it was active prior to the swap.

Preferably said programmable controller is adapted to record a selected history of the changes of state activity, or of changes of specified values of variables and the history of state activity changes is replayed to the user by displaying the appropriate state diagrams graphically, and indicating on them the route taken by the state activity during the recorded operation of the controller through the network formed by the states and their associated transitions.

Preferably in said programmable controller storage is allocated associated with an individual state diagram, in addition to that allocated for storage of history in general, for the purpose of ensuring that at least a minimum amount of history can be stored for the said state diagram, independent of how long it takes to accumulate that history and how much history is generated by other state diagrams during that time.

Preferably in said programmable controller storage is allocated so that, when a certain defined trigger condition occurs, a defined amount of history may be saved so that it will not be overwritten by subsequent events until so allowed by the system user and the said history saved includes history pertaining to events that occurred immediately prior to the trigger event.

To those skilled in the art to which the invention relates, many changes in construction and widely differing embodiments and applications of the invention will suggest themselves without departing from the scope of the invention as defined in the appended claims. The disclosures and the descriptions herein are purely illustrative and are not intended to be in any sense limiting.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is an illustration of a screen shot from a computer running a programming environment according to the preferred form of the present invention with a graphical representation of a function block diagram in the right hand pane,

Figure 2 is an illustration of a screen shot from a computer running the programming environment according to the preferred form of the present invention with

a graphical representation of a state diagram in the middle pane and a corresponding textual representation of the same state diagram in the right hand pane, and

Figure 3 is an illustration of a computer screen shot from a computer running the programming environment according to the preferred form of the present invention with a text listing of state activity, as recorded by the programmable controller, in the right hand pane.

DETAILED DESCRIPTION

IEC 1131-3 specifically states in clause 1.4.3 that "A programming language other than one of those defined in this standard may be used in the declaration of a function or function block". That means that the code defining the operation of a function block can be written in any language. Clause 1.5.1 note (e) describes how extensions of the standard languages are to be documented. We are therefore free to either modify an existing IEC 1131-3 language or use a completely different one for the purpose of programming function blocks. We will however show how we can include State Diagrams with the very minimum modification to the languages. The modifications amount to the addition of a single function block type called a state diagram function block, with possibly but not necessarily also the addition of STATE and GOTO statements dependent on the methods of information input, modification and organisation for the state diagram function block type. The standard leaves the design of these methods up to the designer of the IEC 1131-3 system, providing they are consistent with the IEC 1131-3 system as defined in the standard.

IEC 1131-3 systems as provided in the commercial world do not necessarily comply rigidly with the standard, but rather follow the general approach taught by the standard and implement the essence of the ideas covered by the standard. This invention relates to improvements to such commercially available systems, as well as to completely standard systems.

The invention can be viewed from several different perspectives. One is the form of the code used to provide the capability with minimum added complexity. Another is the software structure used to enable the operation of the system. Another is the manner in which the code and the software structure is presented to the user and entered by the user.

The State Diagram code is incorporated into the system as the code that defines at least part of the operation of a function block in a function block diagram. Each FB encapsulates the code describing one or more state diagrams. In the preferred form of the

invention the function block diagram forms the top level of a program and defines the permitted interactions between one or more state diagrams.

Textual form of state diagram code

In one form of our invention a state diagram may be defined in text in the following general form

```
STATE StateName1
<Statement> \
<Statement>  |
<Statement>  |
.....  |= Block of code
<Statement>  |
<Statement>  |
<Statement> /
END-STATE

STATE StateName2
<
Block of Code
>
END-STATE

STATE StateName3
<
Block of Code
>
END-STATE

STATE StateName4
<
Block of Code
>
END-STATE
Etc.
```

Each state is described by a STATE statement which defines the name of the state, followed by a block of code that defines what processing and control will occur while the state is active. The end of a state is delimited by an END-STATE statement.

Each block of code consists of none, one or more optional statements, which define data processing or control to be performed while the state is active, and the conditions which will lead to changes of state activity together with the identity of the state that will next become active. The former type of statements is referred to as state action statements, and the latter as state transition statements In practice the two types may be combined.

There may be multiple alternative state transition statements in any state defining which state will become active next, providing that only one state actually becomes active at a time.

A state may have no state action statements, in which case it simply performs no actions. A state may also have no state transition statements, in which case it will stay active for all time once it becomes active, at least as far as the program is concerned.

IEC 1131-3 is a system in which the function blocks are cyclically scanned, although provisions are also made for scanning to be performed on a timed or task priority basis. Each time they are scanned the function block code is executed. Normally for any particular function block the piece of code that executes is the same on each scan, although the logic of the code may cause execution to take different paths through the code on different occasions. With code describing a state diagram the compiler and operating system allocate a variable to each function block which executes state diagram code. That variable holds a value identifying which state in the state diagram is active. When a function block is scanned the value is used by the operating system to identify the code to be executed this particular scan, and the code is scanned. No other code associated with the same state diagram is scanned on the same scan except for a few minor instances that are not relevant to this description. Only one state can be active at any one time. When a transition condition becomes true the corresponding GOTO statement is executed, and this alters the value of the variable to indicate the state that is to be active and the code that is to be executed on the next scan of the function block.

Conditional GOTO statements or their functional equivalents therefore define state transitions. The mechanism that ensures that only one state becomes active at a time is as follows. When a GOTO statement executes, it immediately changes the value identifying the active state in the variable provided by the operating system for the purpose. When this is done, scanning of the state ceases, thereby precluding the possibility that additional GOTO statements will be executed. It would also be possible to allow scanning to continue to the end of the block, but this would require a language rule that allowed the

last GOTO statement executed to overwrite the effects of any prior executions, and this is not the preferred implementation.

It is important to note that the variable is at least logically associated with the function block as a property of the function block and its value indicates the active state in the function block's state diagram. The variable does not need to indicate anything about the activity of the other states in the state diagram, as by definition, all the other states must be inactive. Only one state is allowed to be active in a state diagram at a time. In particular, the variable is not, and does not need to be a set of Booleans as may be the case in a Sequential Function Chart system where more than one step may be active at one time, and where the activity of each step is indicated by a separate Boolean in the set. Normally the variable will be a direct indicator of the state that is active, such as a state number. It may beneficially be a pointer or an index into a table of state code locations.

A system may have function blocks that are programmed in terms of state diagrams, and ones that are not. If a system is required to have code emulating the operation of more than one state diagram simultaneously, then this may be accomplished by incorporating more than one state diagram function block.

The system is also easily extended to function blocks that execute more than one state diagram simultaneously.

The code and data structures used to support the operation of state diagram function blocks are fundamentally different from the code and data for standard function block in that they incorporate one or more variables to identify the active states and use them to determine which state code to execute. These functions are automatically provided by the system and do not need to be programmed by the applications programmer who uses the system.

The state variable(s) also supports the recording of state activity so that the operating system may, on user command, display the order in which the states became active and the logical conditions on which the state activity changes depended. Again this history recording is an automatic system function, and although it may be set up to occur in a way designed to best suit the user, the user does not need to program the system to record the necessary data.

Methods of use

It is considered advantageous to provide user interfaces to computers using graphical techniques, and the general methods of so doing are well known. The interfaces are called Graphical User Interfaces or GUIs.

An established practice with GUIs to display parts of the system functionality using an object paradigm. If a system incorporates a temperature control function, one of the interface objects, which the user sees as graphics on the associated computer monitor, may well be a temperature controller. The temperature controller may well have scales that visually indicate actual and set-point temperatures and controls such as buttons which enable the use of a computer mouse to adjust the set point.

We will refer to such things as the temperatures as 'properties' of the object. The properties tell us something about the objects condition such as the set-point is at 110 degrees. We will refer to such things as using the buttons as 'sending messages' to the object. Clicking on a button marked 'Increase Set-point 1 Degree' causes the system to send a message to the temperature controller object to increase the set-point property by 1 degree.

For the purpose of this description, a property is therefore a constant or variable associated with functionality that is conceptually handled as an object, and the property has a value.

In fact, in such a system, no physical temperature controller exists. The system software code and variables provide the temperature control functionality. The graphics simply represent a convenient system model that the user can relate to and use to interact with the system. Internally, the code and variables may or may not be organised using a methodology that reflects the display model.

In this invention, the overall system program is represented as a function block diagram (FBD) and state diagrams (SDs) are represented on the FBD by State Diagram Function Blocks (SDFBs). This is illustrated in Figure 1 with the FBD 1 shown in the right hand pane. The SDFBs are represented by boxes 2 and the permitted interactions between the SDFBs are represented by lines 3. The lines 3 represent inputs and outputs from the respective function blocks, and therefore the transfer of values of variables between function blocks. It should be noted that the variables within each function block can be classed as local variables and it is the value of a variable of one block that is transferred to a variable of another block as indicated by the connections in the FBD.

SDFBs have inputs, outputs and internal variables in the same way as normal function blocks. A SDFB is an object, which during simulated operation may have properties associated with it such as the identifier (name) of the active state on the SD or a red/green status indicator to show whether the SD is stopped/running respectively. A SDFB may also be able to accept messages such as a double click on the red/green indicator, which might toggle the status property from running to stopped or vice versa.

One of the most important properties of a SD is the detail program that it

represents. In the preferred form of the invention a SDFB may be "opened" to display the detail program.

Two alternatives present themselves as to how the display of program may be opened, although other methods may be used. The two being considered here consist of Method 1:

1. Open the SDFB to show the SD associated with the SDFB.
2. Open a state to show the block of code associated with the state. Show the name of the state in the window header of the window displaying the code.
3. Close the SD and display of code when no longer required.
4. Allow multiple windows containing SDs or code to be open at any one time.

Method 2

5. Open the SDFB to show the SD associated with the SDFB.
6. Open a state to show all the code associated with the SD, possibly with the code display centred on the block of code associated with the said state, or the code for the said state otherwise highlighted. Show the name of the state in a STATE statement as illustrated above, and use the END-STATE statement to delimit the blocks of code associated with individual states.
7. Close the SD and display of code when no longer required.
8. Allow multiple windows containing SDs or code to be open at any one time.

The second method above is illustrated in Figure 2 where the state diagram 11 is shown in the middle pane 12 and the corresponding textual code is shown in the right hand pane 13.

The fundamental difference between the two is whether STATE statements as such are required in the code, or whether they can be made completely transparent to the user who would rely solely on the graphical representation in the state diagram. The two methods are compared to show that the inclusion of actual STATE statements in the code is more a matter of choice of display method than of being an essential requirement of the software system. The system with the GUI interface can unambiguously display a state program without the use of STATE statements, although their use may be beneficial.

The same arguments apply to GOTO statements. Code associated with GOTOs can be displayed in separate windows or panes of windows with the current and target state identifiers already defined by the state diagram automatically displayed in some convenient form, and the actual GOTO is then redundant.

The conclusion is that if a way is provided of associating a state identifier with the code associated with the state including GOTO code, and associating GOTO code with

the target state identifier, then that is all that is required to define and implement SD functionality. In a non-preferred form even the graphical SD is unnecessary as the associations required can be established by data entry in some other form, such as for instance entry in a tabular form. This is the software equivalent of providing labelled boxes for sections of code together with an indication of which box is currently the active one.

That is not to say that there are no practical advantages of one method over the other.

As has been suggested above the programmable controller can record the operation of the program by recording the sequence of transitions between states of each state diagrams and by recording for each state diagram the changes in the internal variables used in that state diagram. This record can be transmitted back to the programming work station, if the program is being run on a specialised controller, or can be recorded directly by the programming environment if generated during a simulation or where the program is being run from the workstation.

To assist with debugging the program the recorded sequence can be displayed to a user in textual form, for example in the manner illustrated in Figure 3, where the sequence is set forth in a chart 21 in the right hand pane 22. Even more advantageously the recorded sequence can be replayed to the developer graphically, being shown on the function block and state diagrams, such as in the manner referred to above for simulations.(It should also be noted that this environment is suited for the real time direct running of programs, from the work station if the station is suitably equipped with communications devices, in which case the continuing operation of the program can be displayed in the manner indicated with respect to a simulation or replay).

In general though programs developed in the programming environment will be downloaded to a controller, and the controller will record the sequence of operations. To this end the controller is preferably programmed at an operating system level to reserve memory for the recording of the state sequence and variable changes, and in particular to reserve at least a predefined amount of space for the recording of data in respect of the operation of each state diagram. This space is then to be occupied with the most recently recorded data, so that upon the program entering a failure condition the sequence of state transitions and variable changes immediately prior to the failure will be recorded.

In a further improvement to standard IEC 1131-3 systems, the manner of addition of state functionality provides the opportunity to make program modifications easily while the program is running. Because each state is compiled into its own associated block of compiled code, it is possible to replace the code for a state while the program is being run,

by downloading the new block of compiled code, and changing the value of the variable associating each state with its respective code blocks between scans of that state. In this way the new code will be executed instead of the old code when that state is next called to be scanned.

Essential additions to the IEC 1131-3 system

We have shown that basic state diagram functionality can be added to an IEC 1131-3 system through a number of steps, none of which depart from the spirit of the standard:

1. Provide a special FB type with a variable to identify the active state in at least one encapsulated state diagram.
2. Provide a means of associating specific code with specific states including state action and GOTO code.
3. Provide a means of associating specific GOTO code for specific states with specific target states.
4. When the FB is scanned, use the said variable to identify the action code and GOTO code to be executed, and executing that code

It is of the essence of IEC 1131-3 that it leaves the details of how FBs are programmed and how the program is entered to the system designer.

Advantages

Some of the possibilities of the present invention are as follows:

-It is possible to provide a system to implement a state diagram program for encapsulating within a SDFB without any need to add text statements to any of the text languages of IEC 1131-3. The essential associations described above can be achieved by providing appropriate means and contexts, such as tables, dialog boxes or windows, in which to enter the code. All the program can be provided in one of the IEC 1131-3 forms using nothing more than one of the standard languages.

-It is possible to provide a system to implement a textual form of a state diagram program for encapsulating within a SDFB with only additions that implement the functionality of the STATE and GOTO statements. This can be used for such purposes as transferring programs between systems, or printing programs out for inspection of the code. The complete program can be provided in one of the IEC 1131-3 textual forms using nothing more than one of the standard text languages. This means that the only new learning required of someone familiar with IEC

1131-3 is –

What a SDFB is,

How SDs are used, and

How the SD program preparation and debugging tools are used.

Each of these is a simple issue, particularly because of the GUI that can be provided to support them.

In the present specification "comprise" means "includes or consists of" and "comprising" means "including or consisting of".

The features disclosed in the foregoing description, or the following claims, or the accompanying drawings, expressed in their specific forms or in terms of a means for performing the disclosed function, or a method or process for attaining the disclosed result, as appropriate, may, separately, or in any combination of such features, be utilised for realising the invention in diverse forms thereof.

CLAIMS:

1. Apparatus for assisting a programmer in the programming of a programmable controller, comprising:

means for representing a function block diagram having one or more function blocks represented graphically, each with inputs and/or outputs which inputs or outputs may be connected graphically to outputs or inputs of another said function block,

means for allowing user creation and manipulation of said function blocks and of said input/output connections,

means for representing a state diagram, having one or more states represented graphically, and possible transitions between states represented graphically by connections between states,

means for allowing user creation and manipulation of said states and of said transitions,

means for allowing a user to enter coded instructions in relation to a state, and

means for compiling program code for execution by a control processor from said user created function block diagram, state diagrams and entered instructions.

2. Apparatus as claimed in claim 1, including means for internally associating each said state diagram with a said function block and wherein said compiling means reports an error if said user has entered instructions or a transition between states of a state diagram which requires an input or sets an output which is not represented in the function block diagram as being an input or output from the respective function block.

3. Apparatus as claimed in claim 1, including means for representing graphically one of said states as a starting state.

4. Apparatus as claimed in claim 1, including means for debugging which analyse a recorded history of operation of a control processor programmed with said compiled program code, and include means for displaying state activity changes which indicate on said graphically represented state diagrams the route taken by the state activity during said recorded operation of the control processor through the network formed by the states and the transitions between the states.

5. Apparatus as claimed in claim 1, wherein said means for compiling program

code compiles said code into individually identified code blocks, one block for each state of each said state diagram, and said apparatus includes downloading means for downloading one or more individual blocks into said control processor for the purpose of adding new states or of modifying existing states.

6. Apparatus as claimed in claim 1, wherein said downloading means downloads said code blocks in the period between program scans, and changes the value in an identifying variable indicating the block of code which represents a state in the state diagram, and in this way causing the controller to execute different code for the state when it is active after the program swap than would have been executed prior to the program swap.

7. A programming environment for programming a control processor, said environment generally implementing the methods set forth in the IEC 1131-3 standard, including at least the provision of a graphically representable function block based top-level structure, characterised in that the environment provides a specialised state-diagram function block type, which function block type is adapted to implement one or more state diagrams defined by a user.

8. A programming environment as claimed in claim 7, wherein said function blocks representing state diagrams have an associated state variable for each state diagram, provided by said programming environment, the value of which is indicative of the state that is currently active on the state diagram.

9. A programming environment as claimed in claim 7, wherein said function blocks representing state diagrams have a status variable, provided by said programming environment indicating at least whether said state diagram is running or stopped.

10. A programming environment as claimed in claim 7, wherein said function blocks representing state diagrams have a variable, provided by said programming environment, indicating whether the current scan is the first scan since the current state, as defined by said state variable, has become active since its last period of inactivity.

11. A programming environment as claimed in claim 8, wherein said state variable is a pointer or an index into a table.

12. A programming environment as claimed in claim 8, wherein said state variable is used as part of an automatic operating system function that records the history of state activity as the control process progresses.

13. A programming environment as claimed in claim 8, wherein said state diagram is coded in one of the standard IEC 1131-3 languages with the addition of state and transition defining functionality.

14. A programming environment as claimed in claim 13, wherein the state functionality is included by virtue of the operating system storing state code blocks as individually identifiable items, referenced by said state variable.

15. A programming environment as claimed in claim 14, wherein said programming environment compiles said state diagram related code into individually identifiable blocks, with one said block per state, and said identifiable blocks may be individually loaded into the controlling process for the purpose of adding new states, or modifying existing ones in the period between program scans, and implements the program swap by changing the value in an identifying variable indicating the block of code which represents a state in a state diagram, and in this way causes the controller to execute different code for the state when it is active after the program swap than would have been executed when it was active prior to the swap.

16. A programming environment as claimed in claim 13, wherein said programmable control processor is adapted to record a selected history of the changes of state activity, or of changes of specified values of variables and the history of state activity changes is replayed to the user by displaying the appropriate state diagrams graphically, and indicating on them the route taken by the state activity during the recorded operation of the controller through the network formed by the states and their associated transitions.

17. A programming environment as claimed in claim 16, wherein in said programmable control processor storage is allocated associated with an individual state diagram, in addition to that allocated for storage of history in general, for the purpose of ensuring that at least a minimum amount of history can be stored for the said state diagram, independent of how long it takes to accumulate that history and how much history is generated by other state diagrams during that time.

18. A programming environment as claimed in claim 17, wherein in said programmable controller storage is allocated so that, when a certain defined trigger condition occurs, a defined amount of history may be saved so that it will not be overwritten by subsequent events until so allowed by the system user and the said history saved includes history pertaining to events that occurred immediately prior to the trigger event.

19. An apparatus substantially as hereinbefore described with reference to and as shown in the accompanying drawings.

20. A programming environment substantially as hereinbefore described with reference to and as shown in the accompanying drawings.

21. Any novel feature or combination of features disclosed herein.

PATENTS • DESIGNS • COPYRIGHT • TRADE MARKS
The Patent Office
INVESTOR IN PEOPLE

Application No: GB 9923193.8 **Examiner:** Mike Davis
Claims searched: 1-20 **Date of search:** 26 April 2000

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

UK Cl (Ed.R): G4A (APL)

Int Cl (Ed.7): G06F

Other: Online: WPI, EPODOC, JAPIO

Documents considered to be relevant:

Category	Identity of document and relevant passage	Relevant to claims
A	US 5247693 (BRISTOL) eg abstract	-

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.